MGRIDGEN / PARMGRIDGEN *

Serial/Parallel Library for Generating Coarse Grids for Multigrid Methods

Version 1.0

Irene Moulitsas and George Karypis

University of Minnesota, Department of Computer Science / Army HPC Research Center
Minneapolis, MN 55455

{moulitsa, karypis}@cs.umn.edu

December 5, 2001

 This work was supported by NSF CCR-9972519, EIA-9986042, ACI-9982274, by Army Research Office contract DA/DAAG55-98-1-0441, by the DOE ASCI program, and by Army High Performance Computing Research Center contract number DAAH04-95-C-0008.

Contents

1 Introduction

MGRIDGEN and PARMGRIDGEN are serial and parallel software packages that implement various algorithms for generating a sequence of coarse grids. The generated coarse grids contain well-shaped elements and thus they are well-suited for geometric multigrid methods. PARMGRIDGEN is an MPI-based parallel library that is based on the serial package MGRIDGEN and it is suited for parallel numerical simulations involving large unstructured grids since the algorithms incur a very small communication overhead, achieve high degree of concurrency and maintain the high quality of the coarse grids obtained by the serial algorithms in the MGRIDGEN library.

This manual is organized as follows. In Section 2 we briefly describe the serial algorithms of MGRIDGEN and the parallel approach to the problem of coarse grid construction. In Section 3 we describe the use of the standalone programs, provided with MGRIDGEN and PARMGRIDGEN, to compute coarse grids. Section 4 describes the format of the basic parameters that are supplied to the routines. Section 5 provides a detailed description of the calling sequence of the functions in both libraries. Finally, Section 6 describes what other libraries you need in order to run MGRIDGEN and PARMGRIDGEN and provides contact information.

2 Overview

The goal of the underlying algorithms in MGRIDGEN/PARMGRIDGEN is to generate a coarse grid that contains well shaped elements, subject to the minimum and maximum size constraints on the size of these fused elements. MGRIDGEN/PARMGRIDGEN implement four different methods for measuring the overall quality of the coarse grid, and the particular method can be selected by the user.

Consider a grid G_0 with n elements and let G_1 be the coarse grid with n_1 elements, such that each element of G_1 is obtained by combining together (*i.e.* fusing) some elements of G_0. For each element of the coarse grid we compute its aspect ratio, that measures the degree to which its corresponding area, for two-dimensional grids, or volume for three-dimensional grids, is circular or spherical. For two-dimensional grids, the aspect ratio of an element is defined as $A = \frac{l^2}{s}$. Whereas, for three-dimensional grids, the aspect ratio is defined as $A = \frac{S^{\frac{3}{2}}}{V}$. Within this setting, the four methods for measuring the quality of the coarse grid G_1 are defined as follows :

Sum of Aspect Ratios We determine the overall quality by summing up the aspect ratios over all the elements of G_1. Since smaller aspect ratios are better, a grid that has the smallest sum of the aspect ratio of its elements is preferred.

Weighted Sum of Aspect Ratios We can take into consideration the size of each element in the grid (*i.e.* how many elements of the original grid were fused together to create an element in the coarse grid) and use this information as the weight parameter for each element when computing the weighted sum of aspect ratios.

Maximum Aspect Ratio Another way to measure the quality of the coarse grid is to minimize the aspect ratio of the worst element in the coarse grid. Therefore, a preferred coarse grid will be the one that achieves the best possible aspect ratio for its "worst" element.

Maximum Aspect Ratio and Weighted Sum of Aspect Ratios The third and second quality measures can be easily combined together and used as a new method to measure the quality of the coarse grid. In this case, among grids that have the same aspect ratio for their worst element, the weighted sum of the aspect ratios will be used.

MGRIDGEN/PARMGRIDGEN use the multilevel paradigm to find the coarse grid that optimizes the particular coarse grid quality measure. Within that context, the quality measure function becomes the objective function that is optimized by the multilevel paradigm. The general framework of the multilevel paradigm is shown in Figure 1, that has been found effective in the related problem of graph/mesh partitioning, see [1, 2, 4, 3].

An important aspect in the whole algorithm is the method used to do the coarsening phase of the multilevel algorithm. MGRIDGEN/PARMGRIDGEN implement two methods for the coarsening phase, and again, the particular method can be selected by the user.

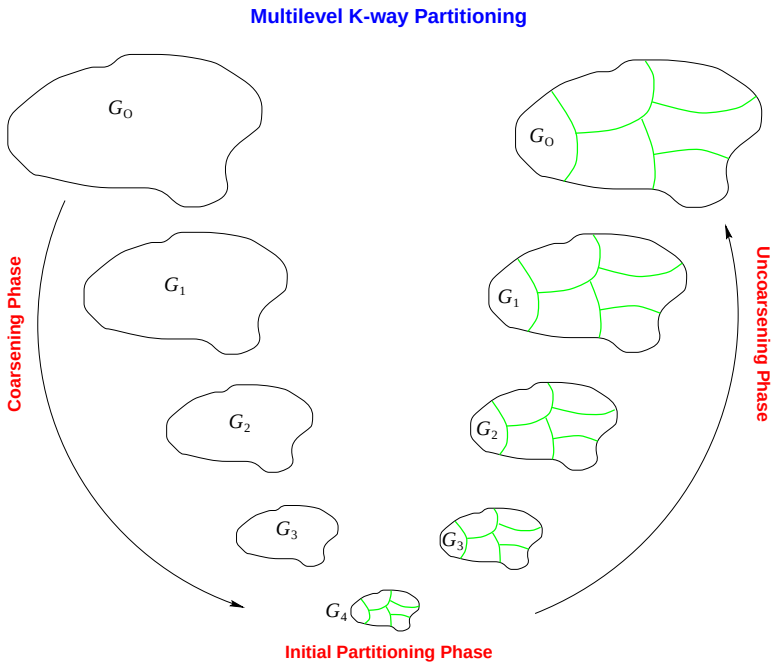


Figure 1: The various phases of the the multilevel graph partitioning.

Random Agglomeration This is a completely randomized algorithm where neighboring elements are fused together in a random way without considering the quality of the induced coarse grid.

Globular Agglomeration In this agglomeration technique neighboring elements are fused together in such a way that the new fused coarse element will have a minimal aspect ratio.

Throughout our algorithms the grid is modeled using its dual graph, where each vertex in the graph corresponds to an element of the grid (*i.e.*, triangle, tetrahedron, brick), and two vertices are connected via an edge, if the corresponding grid elements share a segment or face depending on whether or not the grid is two- or three-dimensional.

Figure 2 gives an overview of the functionality provided by MGRIDGEN/PARMGRIDGEN. Extended tests have shown that certain algorithms tend to produce grids of better quality. These algorithms are the ones appearing in shaded boxes in Figure 2. Here we present all available algorithms in the library but nonetheless we suggest using the ones that perform better. All of our algorithms are described in detail in [6].

2.1 Parallel Formulation

In the parallel formulation of our algorithms, the graph is first distributed among the processors using a parallel graph partitioning algorithm so that well shaped subdomains formulate on each processor such that the number of elements shared on the interface is minimal. Then, every processor modifies its local graph by just removing/cutting the edges that connect it to other processors. Therefore, all processors now have their own subdomain and can work on it, using simply the serial routines provided in MGRIDGEN, without needing to communicate at all with others. The coarse domain obtained after that will have very good characteristics on the inside, but will suffer on the interface since some of the original edges were simply cut. To cure this problem we slightly perturb the original partition of the graph so that previously interface nodes (whose coarsening suffers) become interior in a processor, and formerly interior nodes (whose coarsening is very good and do not need any more work) become interface, see Figure 3. This is done simply by calling an adaptive graph partitioning routine from PARMETIS [5]: ParMETIS_RepartLDiffusion. After the graph is repartitioned between the processors, some local refinement is done to improve the quality of the previously interface nodes. Only a few iterations of repartitioning followed by refinement are needed and we have obtained a coarse grid of the same quality and characteristics as the one that the serial algorithms of MGRIDGEN would yield.

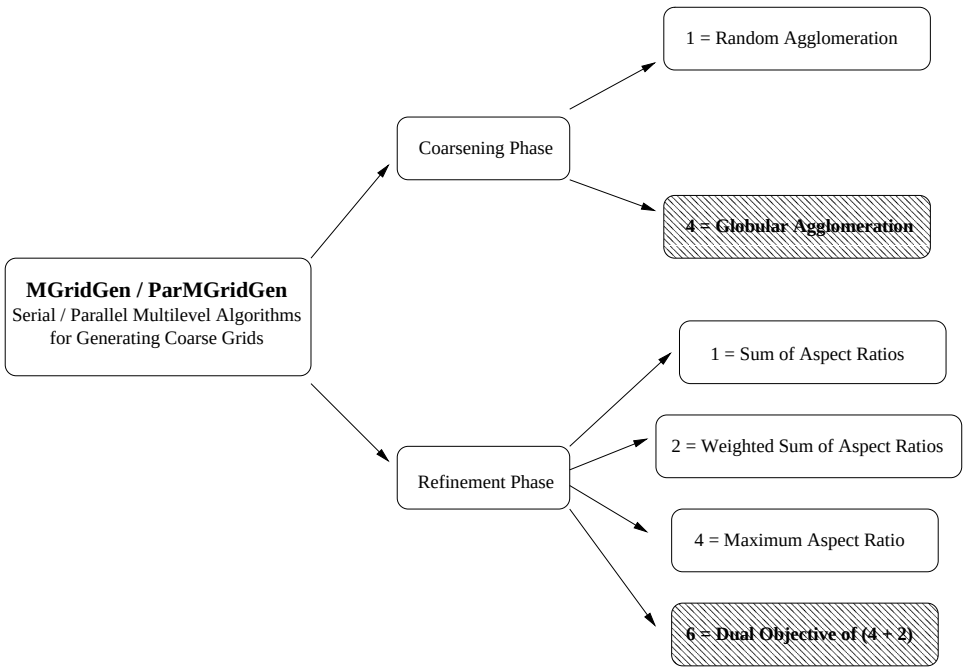


Figure 2: An overview of the functionality provided by MGRIDGEN. Shaded boxes correspond to the algorithms in MGRIDGEN that tend to produce grids of higher quality.

The structure of the parallel algorithm appears in Figure 4.

3 Stand-Alone Programs

MGRIDGEN provides a program, `mgridgen` that can be used to coarsen an unstructured grid. `mgridgen` is invoked by providing the arguments in the command line as follows :

mgridgen $GraphFile\ Dim\ CType\ RType\ LMin\ LMax\ DbgLevel$

Similarly, PARMGRIDGEN provides a program, `parmgridgen` which is invoked by providing the same arguments in the command line. Assuming an installation of MPI is on the machine then the program is invoked as:

mpirun -np $npes$ **parmgridgen** $GraphFile\ Dim\ CType\ RType\ LMin\ LMax\ DbgLevel$

The first argument, $GraphFile$, is the name of the file that stores the grid (whose format is described in detail in Section 4.3). The second argument is the dimension of the problem, either 2 for a two-dimensional grid or 3 for a three-dimensional one. The third and fourth arguments, $CType$ and $RType$, specify which coarsening and refinement algorithms to be used (described in Section 5). The $LMin$ and $LMax$ parameters specify a lower and upper bound, correspondingly, on the size of the induced agglomerated cells. Finally, the last parameter specifies the level of output the user wants to have in the end. Giving $DbgLevel$ of 0 will produce no output information at all. We suggest debug level of 128 which prints some reasonable amount of statistics of the quality of the coarse grid.

Upon successful execution of the serial code the result is stored in file $GraphFile$.part.$nparts$. Similarly, the result of the parallel code is written in file $GraphFile$.part$npes$-$mype$.$nparts$. The format of both files is described in Section 4.3.

4 Input/Output Formats

4.1 Format of the Input Grid

MGRIDGEN takes as input the following information

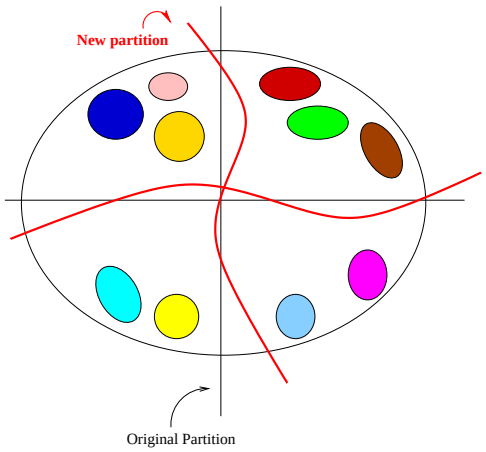


Figure 3: Adaptive Repartition.

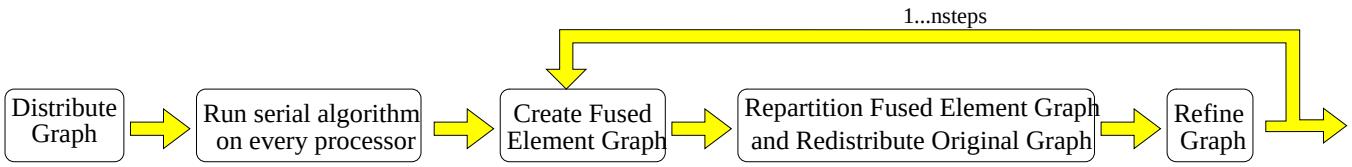


Figure 4: The various phases of the the parallel procedure.

- adjacency structure of the grid in compressed storage format (CSR),
- *vertex-boundary-surface* (v^s),
- *vertex-volume* (v^v)

PARMGRIDGEN takes as input the following information

- adjacency structure of the grid in compressed storage format (CSR),
- *vertex-boundary-surface* (v^s),
- *vertex-volume* (v^v)
- distribution array of the graph,

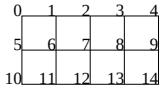
Note that the only difference between the serial and the parallel library is that PARMGRIDGEN requires the distribution of the graph as well.

Serial CSR Format The CSR format is a widely used scheme for storing sparse graphs. In this format the adjacency structure of our graph is represented using two arrays `xadj` and `adjncy`, and the weights on the edges are represented by using one additional array called `adjwgt`. Note that the edge weight here corresponds to either the length of the shared line segment or the area of the shared face, in two- or three-dimensional grids, respectively. Consider a graph with n vertices and m edges. In the CSR format, this graph is stored using the arrays of the following sizes:

$$\texttt{xadj}[n+1], \texttt{adjncy}[2m], \text{ and } \texttt{adjwgt}[2m]$$

Note that the reason both `adjncy` and `adjwgt` are of size $2m$ is because for each edge between vertices v and u we actually store (v, u) as well as (u, v). Also note that in our case the vertices of the graph are unweighted because we assume that they all have the same weight initially. The adjacency structure of the graph is stored as follows.

Assuming that vertex numbering starts from 0 (C style), then the adjacency list of vertex i is stored in array `adjncy` starting at index `xadj[`i`]` and ending at (but not including) index `xadj[`$i+1$`]` (*i.e.*, `adjncy[xadj[`i`]]` through and including `adjncy[xadj[`$i+1$`]-1]`). That is, for each vertex i, its adjacency list is stored in consecutive locations in the array `adjncy`, and the array `xadj` is used to point to where it begins and where it ends. Figure 5(b) illustrates the CSR format for the 15-vertex graph shown in Figure 5(a). The weight of edge `adjncy[`j`]` is stored in `adjwgt[`j`]`. For those familiar with the METIS package, this is exactly the format that is used by the user callable library of METIS.



(a) A sample graph

Description of the graph on a serial computer (MGridGen)

xadj	0 2 5 8 11 13 16 20 24 28 31 33 36 39 42 44
adjncy	1 5 0 2 6 1 3 7 2 4 8 3 9 0 6 10 1 5 7 11 2 6 8 12 3 7 9 13 4 8 14 5 11 6 10 12 7 11 13 8 12 14 9 13

(b) Serial CSR format

Description of the graph on a parallel computer with 3 processors (ParMGridGen)

Processor 0:	xadj	0 2 5 8 11 13
	adjncy	1 5 0 2 6 1 3 7 2 4 8 3 9
	vtxdist	0 5 10 15
Processor 1:	xadj	0 3 7 11 15 18
	adjncy	0 6 10 1 5 7 11 2 6 8 12 3 7 9 13 4 8 14
	vtxdist	0 5 10 15
Processor 2:	xadj	0 2 5 8 11 13
	adjncy	5 11 6 10 12 7 11 13 8 12 14 9 13
	vtxdist	0 5 10 15

(c) Distributed CSR format

Figure 5: An example of the CSR format for storing sparse graphs.

Distributed CSR format The serial CSR format can be extended for the case in which the vertices of the graph and their adjacency lists are distributed among the various processors. In particular, PARMGRIDGEN assumes that each processor P_i stores n_i consecutive vertices of the graph and the corresponding m_i edges, so that $n = \sum_i n_i$, and $m = \sum_i m_i$. Each processor now stores its local part of the graph in the three arrays `xadj[`n_i+1`]`, `adjncy[`$2*m_i$`]`, and `adjwgt[`$2 * m_i$`]`, using the CSR storage scheme. One way of thinking about this distributed CSR format is as follows. Serially you have a single `xadj` and `adjncy` arrays. When you go parallel, the vertices and their adjacency lists are equally distributed among the processors. That is, you can take n/p consecutive adjacency lists from `adjncy` and store them on consecutive processors (p is the number of processors). These portions of the whole `adjncy` array are then the `adjncy` arrays supplied by each of the processors in the distributed CSR format. In addition, each processor supplies its local `xadj` array to point to where each adjacency list begins and ends. Thus, if we take all the local `adjncy` arrays and concatenate them will get exactly the `adjncy` array that is used in the serial CSR. However, concatenating the local `xadj` arrays will not give us the serial `xadj` array, since the entries in each local `xadj` point to their local `adjncy` array.

In addition to these three arrays, each processor also supplies an additional array `vtxdist`$[p+1]$, that indicates the range of vertices stored at each processor. In particular, processor P_i stores the vertices starting from `vtxdist`$[i]$, up to (but not including) vertex `vtxdist`$[i+1]$. Figure 5(c) illustrates the distributed CSR format by an example on a three processor system. The 15-vertex graph in Figure 5(a) is distributed among the processors so that each processor gets 5 vertices and their corresponding adjacency lists. That is, processor 0 gets vertices 0 through 4, processor 1 gets vertices 5 through 9, and processor 2 gets vertices 10 through 14. This figure shows the elements that the arrays `xadj`, `adjncy`, and `vtxdist` store at each processor. Note that the array `vtxdist` will always be the same at each processor.

All four arrays that describe the distributed CSR format are defined in PARMGRIDGEN to be of type either `idxtype` or `realtype`. By default `idxtype` is set to be equivalent to type `int` (*i.e.*, integers) and `realtype` is set to be equivalent to type `double`. However, `idxtype` can be made to be equivalent to a `short int` and `realtype` equivalent to type `float` for certain architectures. The conversion from `int` to `short` and from `double` to `float` can be done very easily by modifying the files `MGridGen/IMlib/IMlib.h` and `ParMGridGen/IMParMetis-2.0/ParMETISLib/struct.h` (instructions are included there). The same `idxtype` is used for the arrays that are used to store the computed partition.

vertex-boundary-surface (v^s) The vertex-boundary-surface array, of length n, is used to capture the length or area of the element's segments or faces that are not shared by other elements, *i.e.*, they are on the boundary of the grid. Note that for all interior elements v^s will be zero.

vertex-volume (v^v) The vertex-volume array, of length n, is used to capture the area (for two-dimensional grids) or the volume (for three-dimensional grids) of that particular element.

An example of a two-dimensional triangular mesh and its corresponding dual graph is shown in Figure 6.

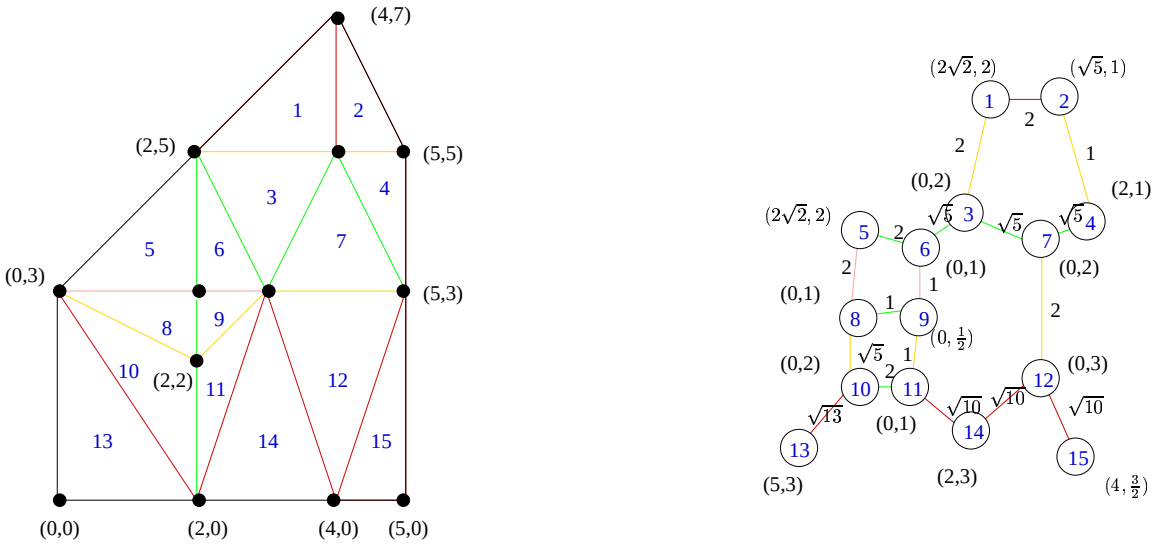


Figure 6: Sample two-dimensional triangular mesh and the dual Graph.

4.2 Output Format of the Computed Coarse Grid

Format of the Partition Array The partition array of a graph with n vertices is an integer array of size n. The ith entry of this array represents the control volume that the ith vertex belongs to. Partition numbers start from 0 up to the number of agglomerated cells (control volumes) produced minus one.

4.3 Format of the Input File for the Stand-Alone Program

The stand-alone PARMGRIDGEN program requires as an input a grid file in the following format. If the grid has n vertices, then the grid is stored in a text file with $n+1$ lines. The first line contains information about the size of the grid, while the remaining n lines contain information for each vertex. The first line contains two integers, which are the number of vertices and edges of the grid, respectively. Note that in determining the number of edges m, an edge between two vertices u, v is counted only one and not twice (i.e., we do not count the edge (u,v) separately from the edge (v,u). For example the graph in Figure 6 contains 18 edges. The remaining n lines contain information about the actual structure of the graph. In particular, the ith line contains information relevant to the ith vertex. In general each line will have the following structure :

$$v^v{}_i \; v_1 \; e_1 \; v_2 \; e_2 \; ... \; v_k \; e_k$$

where $v^v{}_i$ is the vertex-volume associated with the current vertex, v_1, v_2, ... v_k are the vertices adjacent to this vertex and e_1, e_2, ... e_k are the associated weights of these vertices. The vertex-boundary-surface for vertex i is entered simply by entering it as an edge weight between the ith vertex with itself. Note that the vertices here are numbered starting from 1 (not from 0 as is often done in C).

For example, the input file for the grid in Figure 6 would be the one in Table 1.

15	18					
2	1	$2\sqrt{2}$	2	2	3	2
$\sqrt{5}$	1	2	4	1	2	$\sqrt{5}$
2	1	2	7	$\sqrt{5}$	6	$\sqrt{5}$
1	2	1	7	$\sqrt{5}$	4	2
2	6	2	8	2	5	$2\sqrt{2}$
1	3	$\sqrt{5}$	5	2	9	1
2	4	$\sqrt{5}$	3	$\sqrt{5}$	12	2
1	5	2	9	1	10	$\sqrt{5}$
$\frac{1}{2}$	6	1	8	1	11	1
2	8	$\sqrt{5}$	11	2	13	$\sqrt{13}$
1	9	1	10	2	14	$\sqrt{10}$
3	7	2	14	$\sqrt{10}$	15	$\sqrt{10}$
13	10	$\sqrt{13}$	13	5		
3	11	$\sqrt{10}$	14	2	12	$\sqrt{10}$
$\frac{3}{2}$	12	$\sqrt{10}$	15	4		

Table 1: Input Grid File corresponding to grid in Figure 6.

Note here that in the actual file, $\sqrt{5}$ should be substituted by the actual value itself, i.e. 2.236068, etc.

4.4 Output File Format from the Stand-Alone Program

Format of the Partition File The partition file of a graph with n vertices consists of n lines with a single number per line. The ith line of the file represents the control volume that the ith vertex belongs to. Partition numbers start from 0 up to the number of agglomerated cells (control volumes) produced minus one.

4.5 Memory Allocation

MGRIDGEN and PARMGRIDGEN allocate all the memory they requires dynamically. This has the advantage that the user does not have to provide workspace, but if there is not enough memory, the routines abort. Note that the routines in MGRIDGEN and PARMGRIDGEN do not modify the arrays that store the graph.

5 Calling Sequence of the Routines

This is the description of the calling sequence of the routine provided by MGRIDGEN.

MGridGen (int nvtxs, idxtype *xadj, realtype *vvol, realtype *vsurf, idxtype *adjncy, realtype *adjwgt,
int minsize, int maxsize, int *options, int *nmoves, int *nparts , idxtype *part)

Parameters

nvtxs The number of vertices of the graph.

xadj, adjncy
The adjacency structure of the graph as described in Section 4.1.

vvol Information about the volume of each vertex as described in Section 4.1.

vsurf Information about the boundary surface of each vertex. It is zero for all interior elements as described in Section 4.1.

adjwgt Information about the weights of the edges as described in Section 4.1.

minsize A lower bound on the cell size for the coarse graph.

maxsize An upper bound on the cell size of the coarse graph.

options This is an array of 4 integers used to pass parameters for the various phases of the algorithm.

options[0] CType : Selects the algorithm to be used for the coarsening phase of the multilevel algorithm, see Section 2, [6]. Possible values are:

1 A random agglomeration algorithm is used

4 A globular agglomeration algorithm is used **(Suggested)**.

The globular agglomeration scheme yields better results and is therefore recommended.

options[1] RType : Selects the algorithm to be used for the uncoarsening / refinement phase of the multilevel algorithm, see Section 2, [6]. Possible values are:

1 Minimize objective F_1 as described in [6].

2 Minimize objective F_2 as described in [6].

4 Minimize objective F_3 as described in [6].

6 Minimize dual objective F_3 and F_2 as described in [6] **(Suggested)**.

The dual objective yields better results and is therefore recommended.

options[2] Specifies the level of information to be returned during the execution of the algorithm. If set to 0 no information is given. If set to **128** it provides some basic information regarding the quality measures of the coarse graph **(Suggested)**. Additional options for this parameter can be obtained by looking at the end of the file `defs.h`.

options[3] Specifies the dimension of the grid. Possible values are :

2 It is a 2-dimensional grid.

3 It is a 3-dimensional grid.

nmoves Upon successful completion, this variable stores the number of moves that were performed during the refinement phase.

nparts Upon successful completion, this variable stores the number of agglomerated cells in the coarse graph.

part This is an array of size equal to the number of graph vertices. Upon successful completion stores the control volume number that each graph vertex belongs to, in the coarse graph, as (described in Section 4.2).

The calling sequence of the routine provided by PARMGRIDGEN is described in the rest of this section.

ParMGridGen (idxtype *vtxdist, idxtype *xadj, realtype *vvol, realtype *vsurf, idxtype *adjncy, realtype *adjwgt, int *nparts, int minsize, int maxsize, int *options, idxtype *part, MPI_Comm *comm)

Parameters

vtxdist The array describing how the vertices of the graph are distributed among the processors as described in Section 4.1.

xadj, adjncy
The adjacency structure of the graph as described in Section 4.1.

vvol Information about the volume of each vertex as described in Section 4.1.

vsurf Information about the boundary surface of each vertex. It is zero for all interior elements as described in Section 4.1.

adjwgt Information about the weights of the edges as described in Section 4.1.

nparts Upon successful completion, this variable stores the number of agglomerated cells in the coarse graph that is stored locally.

minsize A lower bound on the cell size for the coarse graph.

maxsize An upper bound on the cell size of the coarse graph.

options This is an array of 4 integers used to pass parameters for the various phases of the algorithm. A complete description of this array appeared before in the description of the call of MGridGen.

- options[0] CType : Selects the algorithm to be used for the coarsening phase of the multilevel algorithm.
- options[1] RType : Selects the algorithm to be used for the uncoarsening / refinement phase of the multilevel algorithm.
- options[2] Specifies the level of information to be returned during the execution of the algorithm.
- options[3] Specifies the dimension of the grid.

part This is an array of size equal to the number of locally stored vertices. Upon successful completion stores the control volume number that each graph vertex belongs to, in the coarse graph, as (described in Section 4.2).

comm This is the MPI communicator of the processes that call PARMGRIDGEN. For most programs this will be `MPI_COMM_WORLD`.

6 Hardware & Software Requirements, and Contact Information

MGRIDGEN is written entirely in ANSI C, and is portable on most Unix systems that have an ANSI C compiler (the GNU C compiler will do). It has been extensively tested on AIX, SunOS, Solaris, IRIX, Linux, FreeBSD, and Unicos.

PARMGRIDGEN is written entirely in ANSI C, and uses MPI for communication among the processors. It is portable on most Unix systems that have an ANSI C compiler (the GNU C compiler will do) and MPI. It has been extensively tested on AIX, IRIX, Linux, and Unicos.

Instructions on how to build the libraries are available in the file called `INSTALL` in PARMGRIDGEN's distribution. In order to compile PARMGRIDGEN, the serial code of MGRIDGEN must have been installed on the system first. The distribution of PARMGRIDGEN contains the serial library as well and it is built automatically. In order to use the libraries in your application, you need to link your program with either `libmgrid.a`, or both `libmgrid.a` and `libparmgrid.a`, depending on your application, whether it is serial or parallel.

In the directory called `Graphs` you will find sample graph files, to be used with the standalone programs `mgridgen` and `parmgridgen`, that test if MGRIDGEN and PARMGRIDGEN were built correctly. Also, two header files

called `mgridgen.h` and `parmgridgen.h` are provided that contain the prototypes for the functions in both libraries.

Even though MGRIDGEN and PARMGRIDGEN contain no known bugs, it does not mean that all of their bugs have been found and fixed. If you find any problems, please send an email to *moulitsa@cs.umn.edu*, with a brief description of the problem you have found. Any future updates to MGRIDGEN and PARMGRIDGEN will be made available on WWW at *http://www.cs.umn.edu/~moulitsa/software.html*.

References

[1] T. N. Bui, S. Chaudhuri, F. T. Leighton, and M. Sipser. Graph bisection algorithms with good average case behavior. *Combinatorica*, 7(2):171–191, 1987.

[2] Bruce Hendrickson and Robert Leland. A multilevel algorithm for partitioning graphs. Technical Report SAND93-1301, Sandia National Laboratories, 1993.

[3] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48(1):96–129, 1998. Also available on WWW at URL http://www.cs.umn.edu/~karypis.

[4] G. Karypis and V. Kumar. A fast and highly quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1), 1999. Also available on WWW at URL http://www.cs.umn.edu/~karypis. A short version appears in Intl. Conf. on Parallel Processing 1995.

[5] G. Karypis, Kirk Schloegel, and V. Kumar. PARMETIS 2.0: Parallel graph partitioning and sparse matrix ordering library. Technical report, Department of Computer Science, University of Minnesota, 1998. Available on the WWW at URL *http://www.cs.umn.edu/~metis*.

[6] Irene Moulitsas and George Karypis. Multilevel algorithms for generating coarse grids for multigrid methods. In *Supercomputing 2001 Conference Proceedings*, 2001.